

DOCUMENT RESUME

ED 237 054

IR 010 878

AUTHOR Soloway, Elliot; And Others
TITLE MENO-II: An AI-Based Programming Tutor.
INSTITUTION Yale Univ., New Haven, Conn. Dept. of Computer Science.
SPONS AGENCY Office of Naval Research, Arlington, Va. Personnel and Training Research Programs Office.
REPORT NO YaleU/CSD/RR#258
PUB DATE Dec 82
CONTRACT N00014-82-K-0714
NOTE 39p.; For related documents, see IR 010 877 and IR 010 879.
PUB TYPE Reports - Descriptive (141)
EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS *Artificial Intelligence; *Computer Assisted Instruction; Computer Programs; Computer Science Education; Design Requirements; *Error Patterns; Higher Education; *Human Factors Engineering; *Programming; Systems Development
IDENTIFIERS *Debugging Aids; MENO II; PASCAL Programming Language; Tutorial Mode

ABSTRACT

This report examines the features and performance of the BUG-FINDING component of MENO-II, a computer-based tutor for beginning PASCAL programming students. A discussion of the use of artificial intelligence techniques is followed by a summary of the system status and objectives. The two main components of MENO-II are described, beginning with the BUG-FINDER, which draws on a database of 18 common bug or error types, represented as templates, and attempts to match these templates against its analysis of a student's program. The TUTOR component then infers the student's misconception from the type of error identified. Examples of TUTOR's analyses and interactions are presented. A preliminary evaluation of the BUG-FINDING component used the first syntactically-correct, looping program produced by 20 students randomly selected from 116 volunteers in an introductory PASCAL programming class. Results indicate that the BUG-FINDER found only 22% of the errors and was correct 55% of the time in its analysis. Examples of both correct and incorrect analyses are examined, and reasons for the poor performance of the system are suggested. Plans to extend the MENO-II are discussed, and 25 references are listed. (LMM)

* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *

ED0237054

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION
CENTER (ERIC)

* This document has been reproduced as
received from the person or organization
originating it.

Minor changes have been made to improve
reproduction quality.

- Points of view or opinions stated in this docu-
ment do not necessarily represent official NIE
position or policy.

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE



MENO-II: An AI-Based Programming Tutor

Elliot Soloway, Eric Rubin,
Beverly Woolf, Jeffrey Bonar, W. Lewis Johnson

YaleU/CSD/RR #258

December 1982

IR010878

"PERMISSION TO REPRODUCE THIS
MATERIAL HAS BEEN GRANTED BY

Elliot Soloway

TO THE EDUCATIONAL RESOURCES
INFORMATION CENTER (ERIC)."

*Department of Computer Science
Yale University
P.O. Box 2158
New Haven, Connecticut 06520

**Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

MENO-II: An AI-Based Programming Tutor

December 1982

Research Report #258

Elliot Soloway*, Eric Rubin**,
Beverly Woolf**, Jeffrey Bonar**, W. Lewis Johnson*

IR 010.878
This work was supported in part by the Army Research Institute for the Behavioral and Social Sciences, under ARI Grant No. MDA903-80-C-0508. This work also supported in part by a grant co-sponsored by the Personnel and Training Research Groups, Psychological Sciences Division, Office of Naval Research and the Army Research Institute for the Behavioral and Social Sciences, under Contract No. N00014-82-K-0714, Contract Authority Identification Number, Nr 154-492. Approved for public release; distribution unlimited. Reproduction in whole or part is permitted for any purpose of the United States Government.

Abstract

MENO-II is a computer-based tutor intended to help novices learning to program in Pascal. The BUG-FINDing component attempts to find non-syntactic bugs in a student's program. It draws on a database of 18 common bug types, represented as templates, and attempts to match these templates against its analysis of the student's program. The TUTORing component then attempts to infer the misconception that might underlie the bug and present the student with remedial instruction. We tested the BUG-FINDing component in a classroom setting; we report here on an analysis of the system's performance.

1. Introduction

We all know -- and grumble -- about the error messages that one gets from a computer system. That problem is magnified for the novice programmer who is learning a first programming language. With the number of individuals who are learning programming growing rapidly, there is a clear need for intelligent tutoring systems which can assist the novice at a most critical time: when he/she is alone, one-on-one, with the beast.

2. Using Artificial Intelligence Techniques

There are two major differences between MENO-II¹ and typical frame-based CAI systems:

1. In contrast to more classical CAI systems which build the subject matter into frames which are tied together with a branching strategy, the knowledge about programming which MENO-II has is represented explicitly in the form of a network, and the instructional strategy is independent of that knowledge and also encoded explicitly. Thus, MENO-II generates what to say based on the particular situation.
2. In contrast to more classical CAI systems which respond to only a small set of student errors, MENO-II can cope with 18 different types of program bugs. These bugs are tied explicitly to a knowledge base of potential misconceptions, which is accessed when interacting with the student. While frame-based CAI systems could include more types of bugs, they typically do not do so. It is the recognition that a teaching system needs to handle significant variability in student responses that is the key.

In order to build a system such as MENO-II, we have used many techniques which have been developed in Artificial Intelligence. For example, the knowledge representation we use is called

¹The system's name, MENO, is the name of one of Plato's dialogues in which the question of how learning can possibly take place is discussed. In particular, Socrates questions how a slave-boy can possibly learn the proof of the Pythagorean Theorem: if the boy didn't know it already, then how could he possibly recognize it when it is being taught, but if the boy knew it already then the boy hadn't learned anything. This conundrum is called Plato's Learning Paradox.

KL-ONE, and was developed as a general knowledge representation language [3]. See Clancey [5] and Goldstein [8] for a more extensive discussion of the role which AI can play in CAI.

3. Objectives and Status of System

MENO-II is a tutoring system designed to help novice programmers learn Pascal. The goals of MENO-II are:

- catch *run-time* (semantic and pragmatic) bugs in the student programs²
 - ▶ focus on introductory programs: straight line, branching, simple looping.
 - ▶ use MENO-II in conjunction with an existing lecture course on Pascal.
- suggest misconceptions in the students' heads which underlie the bugs
 - ▶ "talk" in language which is close to programming so student will understand
- instruct/tutor the student with respect to the misconceptions.

The current status of the system corresponding to these objectives is:

- MENO-II can find 18 bugs with respect to repetition and assignment
- MENO-II can suggest underlying misconceptions for those bugs
- only a rudimentary form of tutoring has been implemented in MENO-II.

MENO-II is divided into two major components: The BUG-FINDER, and The TUTOR; each will be discussed in turn. We will also present a preliminary report on an evaluation of the BUG-FINDING component of MENO-II in a classroom setting. We close with a discussion of plans for MENO-II's expansion.

Before proceeding with the detailed description of MENO-II, it might be instructive to compare the goals of MENO-II to those of two other AI-CAI programming tutors, BIP [25] and SPADE-0 [13]. The former system was designed and built to be a self-contained, full course in the programming language BASIC; it was extensively used in a instructional setting. It had a sophisticated technique for deciding what material should be presented to the student, and it had excellent graphic displays. BIP analysed student's programs by running them on test data. In contrast, MENO-II's goals are clearly more limited (e.g., it should simply provide on-line help debugging programs). However, significant effort has been expended in MENO-II to equip it with the means to understand buggy programs, and infer the misconceptions which might underlie the bugs; we report on the limited success of this enterprise later in this paper. SPADE-0 was

²While syntax errors are clearly troublesome for novices, we believe that systems like structure editors (e.g. the Cornell Program Synthesizer [23]) will soon be available, and will help facilitate the creation of syntactically correct programs.

designed, and a prototype built, to teach students to write simple LOGO programs. Unlike either BIP or MENO-II, SPADE-0 forced the student to make the processes involved in programming (e.g., design, coding, debugging) explicit; a student entered a code only after he had provided reasons for why the code was there. Experiments with a successor to SPADE-0 are described in [14].

4. Leverage on Program Understanding: The BUG-FINDER

The BUG-FINDER must be able to recognize two types of bugs: problem independent ones (semantic bugs) and problem dependent ones (pragmatic bugs). An example of the former type is the explicit inclusion of an increment to the index variable in a FOR loop. Bugs of this sort typically reflect confusion about the semantics of the various programming language constructs.

In order to recognize problem dependent bugs, however, the BUG-FINDER must be told what the program is supposed to do. Oftentimes a student's program will run, but it will solve the "wrong" problem. For example, in Figure 1a, we depict a problem for which the the program in Figure 1b is the correct solution; the program in Figure 1c does not solve the problem, but may in fact execute. In other programming tutors, the student provided a specification of the goals of the program (e.g., [6, 13]). In contrast, our approach is to have the teacher provide the specification, since everyone in the course will be working on the same problem.

Currently, the specification of the intended program is represented as a database of *programming plans* that correctly solve the problem. A programming plan reflects stereotypic action sequences in programming (see below). We have also put together and built into the BUG-FINDER a catalogue of common bugs found in simple looping programs. The Bug Catalogue was derived from empirical studies carried out with novice programmers (e.g., [19, 20, 18]). In particular, in one set of studies, we asked students to write programs to solve various problems. In a second type of study, we captured a copy of every syntactically correct program that a student produced while at the terminal [2]. Given the substantial number of programs collected, the systematic analysis of these data are only now being completed [10, 9]. However, observations from these data did help us in compiling the Bug Catalogue.

The BUG-FINDER, which is written in Pascal works by first analyzing a student's program into a "deep structure" representation [4], and then matching that analysis against the bugs in the Bug Catalogue. The deep structure representation of the program specifies the functional characteristics of the program. The set of primitives of this deep structure representation is based on what an expert programmer might know about problems of this type. In particular, this knowledge is focused on types of *looping plans* and the various *roles* which variables play in

(a)

Problem: Read in numbers, taking their sum, until the number 99999 is seen. Report the sum.
Do not include the final 99999 in the sum.

(b)

```
PROGRAM CORRECT-EXAMPLE(INPUT, OUTPUT);  
  VAR SUM, NEW : REAL;  
  BEGIN  
    SUM:=0;  
    READ(NEW);  
    WHILE NEW <> 99999 DO  
      BEGIN  
        SUM:=SUM + NEW;  
        READ(NEW)  
      END;  
    WRITELN(SUM)  
  END;
```

A correct version of a program which solves the above problem.

(c)

A buggy program attempting to solve the above problem. The variable SUM is not initialized to 0. Similarly, the variable NEW has no value the first time it is tested at the top of the WHILE loop. Finally, notice that this program will add the final 99999 to SUM.

```
PROGRAM BUGGY-EXAMPLE  
  VAR SUM, NEW : REAL;  
  BEGIN  
    WHILE NEW <> 99999 DO  
      BEGIN  
        READ(NEW);  
        SUM := SUM + NEW  
      END  
    WRITELN(SUM);  
  END;
```

Figure 1: A Problem, A Correct Solution, and a Buggy Solution

programs. For example, the program in Figure 1b illustrates what we call the New-Value Controlled Running-Total Loop Plan; it is just a special case of the Running-Total Loop Plan, in which the loop, while accumulating a total, is controlled by the value of the Read Variable (i.e., the New-Value Variable). This knowledge is described at greater length in [20], and draws on the work of [13, 15, 16, 24, 17, 1].

We will describe the four stages in the bug finding process in the context of an example. In particular, we will analyze the program in Figure 1c. This program is an attempt to solve the problem in Figure 1a. (A correct version of that program is given in Figure 1b.) Notice that there are several bugs in this program. The variable SUM is not initialized to 0. Similarly, the variable NEW has no value the first time it is tested at the top of the WHILE loop. More subtly, notice that this program will incorrectly add the 99999 into the final sum. This occurs because once in the loop, the value of NEW is first READ, then summed, and only then tested for equality to 99999; thus, by the time the test is made the 99999 has already been added to SUM.

During the first stage of the bug finding process, the program is parsed into an augmented parse tree. (Figure 2 contains the important fragment from the parse tree produced during this stage.) This representation makes it easy to automatically determine the types of expressions and statements, their execution-time sequence, and any nesting of statements (i.e. statements in loops or in an IF statement). All occurrences of each variable are linked together (see Figure 2); since many bugs deal with the use (and misuse) of variables, fast access to the occurrences of variables is needed in order to facilitate subsequent bug analysis.

The next step is to annotate the parse tree with useful information about the various nodes. This information is designed to simplify and summarize parts of the tree for subsequent bug finding steps. For example, the assignment statement

SUM := SUM + NEW

would be annotated as a "running total assignment". In Figure 3, we show the annotations for the parse tree of Figure 2. The annotations are: (1) the WHILE loop is testing the variable NEW; (2) the READ statement is reading into the variable NEW; and (3) the assignment inside the loop assigns a running total to the variable SUM.

During the third stage of processing, the BUG-FINDER searches for instances of the various programming plans. This is a pattern matching process which uses the annotations in the parse tree as feature detectors against which to compare the plans. For example, the features of a Running-Total Variable Plan are, roughly speaking: (1) a variable which is continually updated by (2) a new value which is generated each time through the loop. The assignment statement

SUM := SUM + NEW

Figure 2: Stage 1— Minimally Augmented Parse Tree

A parse tree for a fragment of the program in Figure 1c. The normal parse tree has been augmented with additional links that tie together all occurrences of a variable.

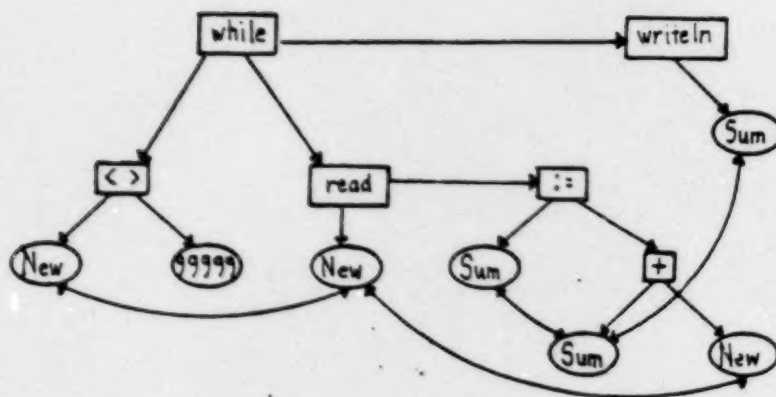


Figure 3: Stage 2-- Summary Annotations Inserted into Parse Tree

The same parse tree as in Figure 2, except that the nodes of the tree have been annotated with information that summarizes the action in the various subtrees.

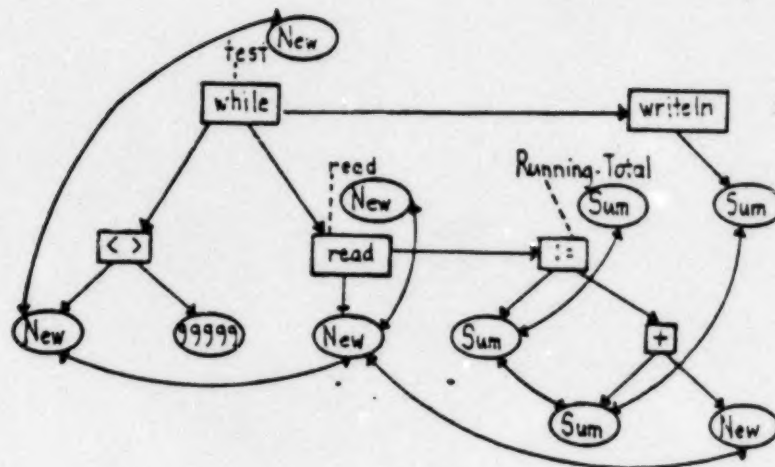


Figure 4: Stage 3— Adding Plan Information to Parse Tree

Plans are identified in the program and this information is added to the parse tree.

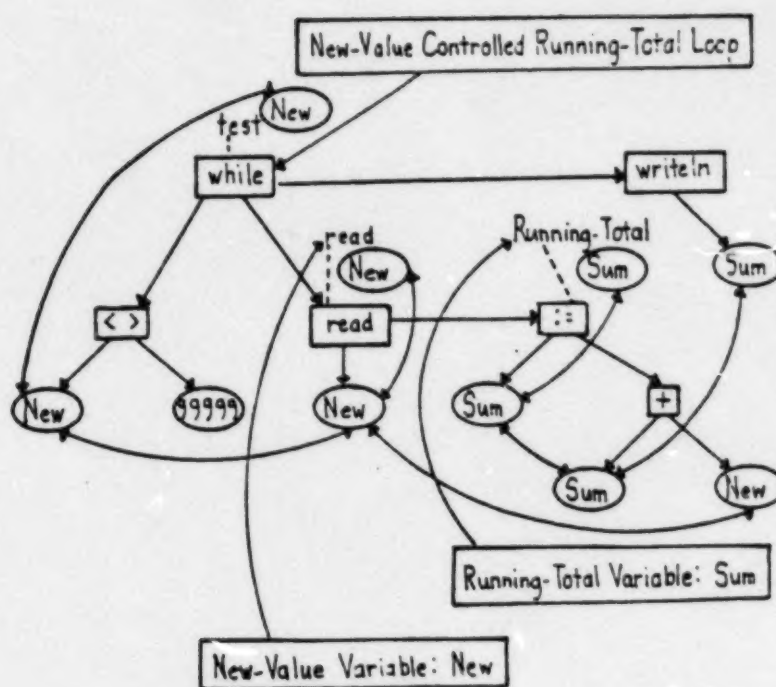
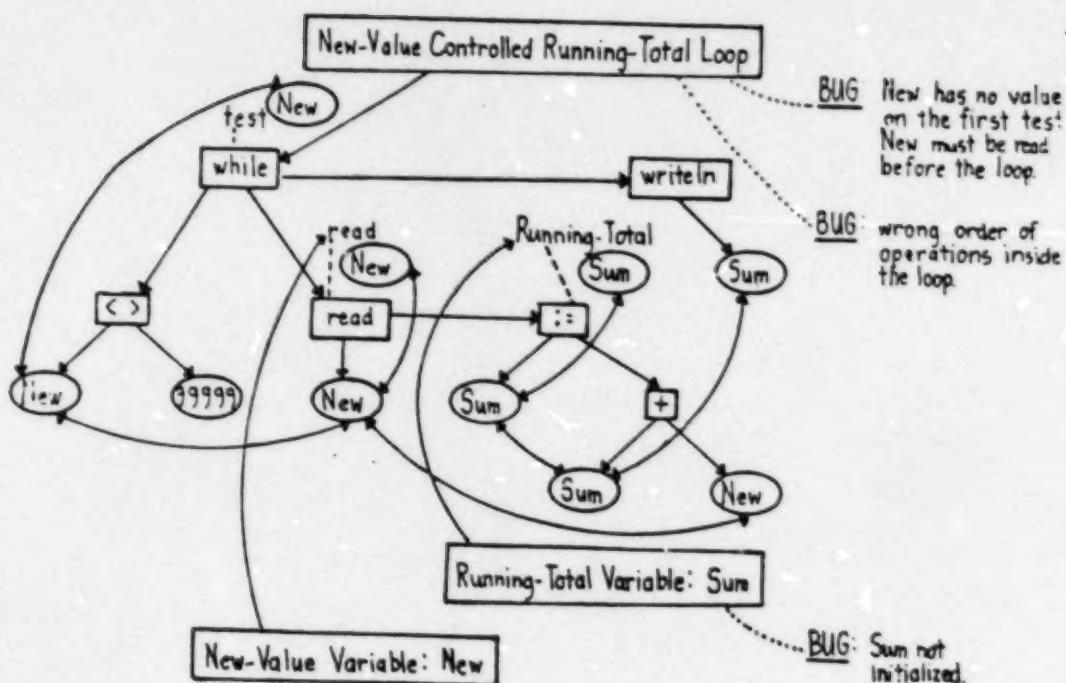


Figure 5: Stage 4— Finding Bugs in the Parse Tree

Finally, bugs are found in the program using the plan annotations in the parse tree and the catalog of common bugs built into the BUG-FINDER.



fits this description, and thus SUM is inferred to be the Running-Total Variable. Similar arguments can be given for the other variable plans and the loop plan. In Figure 4, we show three plans found in the example buggy program: NEW is a New-Value Variable, and SUM is a Running-Total Variable, and the loop is an instance of a New-Value Controlled Running-Total Loop Plan.

Finally, given the augmented parse tree, the annotation, and instantiated plans, the BUG-FINDER program searches the Bug Catalogue for matches. Bugs are detected in terms of the high level information derived from earlier phases. One class of bugs involves variable plans which could not be fully matched, e.g., failure to initialize variable. In Figure 5, the Running-Total Variable, SUM, has such a bug. Other bugs are caused by incorrect ordering of the operations in the loop. For example, the program in Figure 1c will incorrectly add the sentinel value 99999 into the running total; this is a classic example of the "off by one" bug. Moreover, upon entry into the loop, NEW is undefined. A correct version of this program (Figure 1b) would READ a value into NEW before the loop, and would perform the READ on NEW in the loop after the running total update. As indicated in Figure 5, the BUG-FINDER identified these errors in the buggy program.

5. The TUTOR: Inferring Misconceptions from Bugs

Given that the BUG-FINDER has found a bug (or bugs) in a student's program, the next step is to hypothesize the set of potential misconceptions which were in the head of the student which might have been responsible for the program bug. In particular, the BUG-FINDER passes the number of the bug to the TUTOR, and it is the job of the TUTOR to perform this misconception analysis. Currently, the TUTOR hypothesizes the misconceptions and simply reports them, plus the correct concepts, to the student; the TUTOR does not attempt to engage in a dialogue with the student.

There are four knowledge bases currently in the TUTOR.

The Expert Knowledge Model -- the correct knowledge about programming is contained in this component.

The Bug Network -- the common bugs we have identified in our empirical work are catalogued in this component.

The Misconception Network -- associated with a bug(s) are misconceptions which could give rise to the bug; this network explicitly stores the misconception(s) and a tutorial associated with each bug.

The Student Model -- this is the system's hypothesis as to what the student does and does not know, the student's history of interaction, etc.

In what follows we will illustrate how the above knowledge bases are used by the TUTOR to produce its misconception analysis and tutorial output.

6. Examples of the TUTOR's Analyses and Interactions

The TUTOR, which is written in LISP, currently has two modes of operation: Full Comment or Partial Comment. In the latter mode, only errors which the TUTOR deems as "serious" will cause the TUTOR to say something to the student; the objective here was to be less intrusive. In the former mode, any semantic or pragmatic bug will elicit a response from the TUTOR. In the following examples we will illustrate both modes.

Consider first the tutoring session depicted in Figure 7, which is an actual student program. In particular, this program was an attempt to solve the problem in Figure 6; a correct program to this problem is also given. While the correct program reads a value into the New-Value Variable (NEW in Figure 6), the program in Figure 7 instead increments the New-Value Variable (POSIDEN) by 1. Our empirical studies have shown this to be a typical bug.

The TUTOR first prints out (Figure 7 its analysis of the roles which the variables play in the program, e.g., POSIDEN is the New-Value Variable, COUNT is the Counter Variable, etc. Next, both the bug and the correct action are described to the student. Bugs, indexed by their numbers, are stored in the Bug Network, and are tied to the Expert Knowledge Model by Buggy-Version links. For example, in Figure 8 we see that Bug 205 is associated with the New-Value Variable. The notation indicates that the correct way to initialize and update a New-Value Variable is via a read, and that Bug 205 indicates that the update was actually accomplished via an assignment statement. The English descriptions are in part generated in real-time from the networks themselves, and are also in part constructed from canned messages hardwired into the networks.

Associated with a bug is a set of possible misconceptions which the student might have and which could cause the observed bug. This information is stored in the Misconception Network. In Figure 8 we see that there are two possible misconceptions associated with Bug 205 (the bug in the program in Figure 7): the Read Declaration Misconception and the Over-generalize Counter Misconception. The pre-specified text associated with each one of these misconceptions is then displayed to the student. In the next version of the TUTOR, a series of questions will be generated and asked of the the student, in order to help differentiate between the competing hypotheses. Currently the Student Model is simply a record of the bugs and misconceptions constructed by MENO-II. Eventually, more history will need to be kept and used in the diagnosis of a student's misconceptions.

Problem: Read in a set of integers and print out their average. Stop reading numbers when the number 99999 is seen. Do NOT include the 99999 in the average.

```
1 PROGRAM CORRECT-EXAMPLE(INPUT,OUTPUT);
2   VAR
3     TOTAL,NEW,COUNT:INTEGER;
4     AVE:REAL;
5   BEGIN
6     TOTAL:=0;
7     COUNT:=0;
8     READ(NEW);
9     WHILE NEW <> 9999 DO
10      BEGIN
11        SUM := SUM + NEW;
12        COUNT:=COUNT+1;
13        READ(NEW)
14      END;
15     AVE:=TOTAL/COUNT;
16     WRITELN('THE AVERAGE IS ',AVE)
17   END.
```

Figure 6: A Problem and Its Program Solution

```

1 PROGRAM AVERAGE1(INPUT,OUTPUT);
2   VAR
3     SUM, POSIDEN, COUNT: INTEGER;
4     AVE: REAL;
5   BEGIN
6     SUM:=0;
7     COUNT:=0;
8     READ(POSIDEN);
9     WHILE POSIDEN<>9999 DO
10      BEGIN
11        SUM := SUM + POSIDEN;
12        COUNT:=COUNT+1;
13        POSIDEN := POSIDEN + 1;
14      END;
15    AVE:=SUM/COUNT;
16    WRITELN('THE AVERAGE IS ',AVE)
17  END.

```

POSIDEN is the New Value Variable
 COUNT is the Counter Variable
 SUM is the Running Total Variable

You modified POSIDEN by adding POSIDEN to 1 where as...

you should modify the New Value Variable by calling the READ procedure: READ(POSIDEN).

Two misconceptions can be associated with this bug:

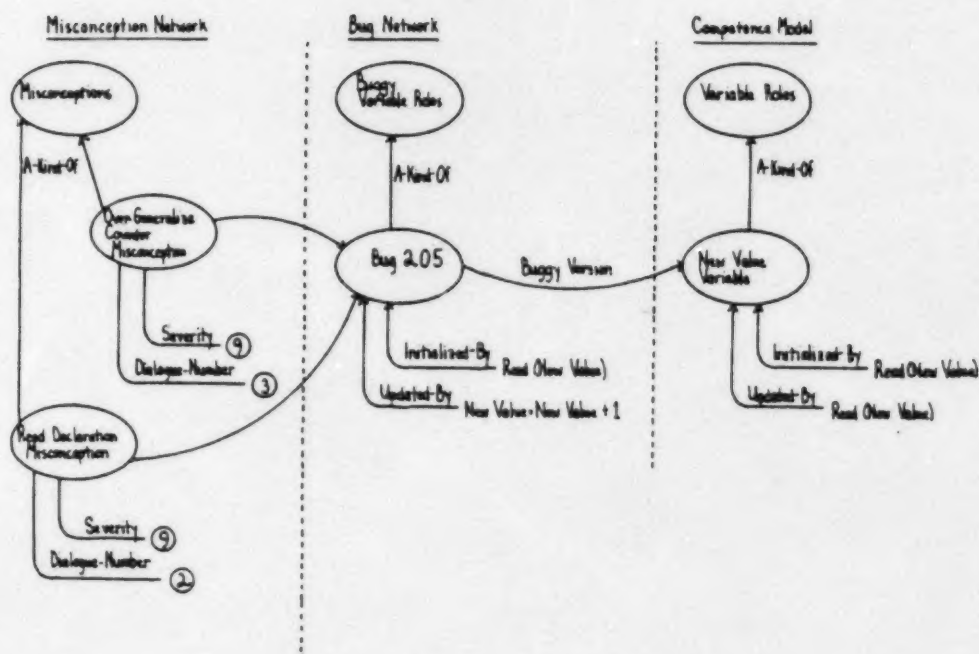
1. You might be thinking that the single call to the READ procedure [READ (POSIDEN)] at the top of your program is enough to define a variable which will always be read in from the terminal. In fact you need to call the read function a second time in your program to read in additional values.

2. You might be thinking that POSIDEN is like COUNT, in that adding 1 to a variable will retrieve its next value. The computer does not know to reinterpret + 1 in the former case to be a READ.

Figure 7: An Example of TUTOR's Analysis

Figure 8: A Slice Through Three Knowledge Bases

The nodes and arcs in this figure represent a small fragment of the actual knowledge bases. Also, the notation has been simplified to enhance readability.



Let us now examine another student program in which there are multiple bugs and multiple possible misconceptions for those bugs. The program in Figure 9 is an attempt at solving a variant of the problem in Figure 6; instead of terminating when the number 99999 is read in, the program should terminate after 10 numbers are read in. There are two types of bugs in this program. The first revolves around the use of the FOR loop. The index variable in the loop, COUNT, is both explicitly initialized before the loop (Bug 1) and explicitly updated in the loop (Bug 2). The former bug is not serious; however, in conjunction with the latter bug, there is strong evidence that the student does not understand that semantics of the FOR loop with respect to its automatic actions on the index variable. Alternatively, the student might be confusing the FOR loop with the WHILE or REPEAT loops, which do require explicit manipulation of the index variable.

The second type of bug in this program is called a "Fractured Assignment Statement Bug" (Bug 3). The objective is to calculate a running total in the variable SUM. Typically, one would write $SUM := SUM + HERMES$. However, the student has introduced an intermediate variable, TEMP, and effected this calculation with 2 assignment statements. We feel that the student is confusing his understanding of how equals works in algebra with how the assignment statement works in programming. Namely, in algebra one would typically feel uncomfortable putting the same variable (SUM in this case) on both sides of the equals sign. Notice that this bug does not lead to an incorrect program. In fact, if the TUTOR were running in Partial Comment Mode, it would not say anything to the student about this bug (assuming of course that this bug was the only bug). The commentary which the TUTOR displays to the student for this misconception is depicted in Figure 9.

7. Testing the BUG-FINDER in the Classroom

7.1. Context of the Study

As discussed in the previous section, MENO-II is made up of a BUG-FINDing component and a TUTORing component. In this chapter, we will describe the results of a study in which the BUG-FINDER was used in an actual classroom setting.

Initially both system components were written in LISP on a VAX 780; this facilitated rapid development. However, we recoded the BUG-FINDER into Pascal since the Cyber 175, on which the actual study would be carried out, did not have a LISP system sufficiently robust for our needs. We did not, however, recode the TUTORing component. We felt a test of the BUG-FINDER's abilities would be very useful by itself, and that we need not wait for the existence of

```

1 PROGRAM AVERAGE2(INPUT,OUTPUT).
2 VAR
3   COUNT,SUM,APPOLO INTEGER,
4   AVE REAL,
5 BEGIN
6   COUNT =1
7   SUM =0
8   FOR COUNT =1 TO 10 DO
9     BEGIN
10      READ(HERMES)
11      COUNT = COUNT + 1,
12      TEMP = SUM + HERMES
13      SUM = TEMP
14    END
15   AVE =SUM/COUNT
16   WRITELN('AVERAGE IS ',AVE)
17 END

```

BUG 1 You have initialized the control variable before the loop. COUNT =1

BUG 2 You have also modified the control variable in the middle of the loop. COUNT = COUNT + 1

Whereas

The FOR loop does an implicit initialization and modification of its control variable. You don't have to mention COUNT at all before or in the middle of the loop

Two misconceptions can be associated with these bugs

1 You might not understand how the FOR loop works. The index variable is initialized automatically, it is updated automatically, and it is compared with the test value for termination

2 You might be confusing the FOR loop with the WHILE loop with respect to the initialization and update of the index variable. The FOR loop does that automatically, while the WHILE loop requires that the user explicitly initialize and update it

BUG 3 It was not necessary for you to write two statements to get the running total SUM

Whereas

You should modify the running total variable by assigning the sum to the running sum variable. SUM = SUM + HERMES

You might be thinking that assignment statements behave like algebra statements. That is you think that $x = a + b$ is equivalent to writing $x = a + b$. However, these expressions are fundamentally different. The assignment statement places a computed value (on the right hand side) into a variable name (on the left hand side). Thus an expression like $x = x + 1$ though meaningless in algebra, is accurate and fairly standard in programming. It places the value of x incremented by 1 into the variable slot.

Figure 9: Multiple Bugs and Multiple Misconceptions

the TUTORing component. Thus, we compiled into the BUG-FINDER much of the TUTORing component's knowledge. In effect then, the BUG-FINDER became a "smart compiler"; it could simply print out a error message about the bug and the potential underlying misconceptions, but it could not engage in any form of dialogue with the student.

The design of the study is as follows. In the fall semester of 1981, we asked students enrolled in an introductory Pascal programming course, to volunteer for our study. We explained that we would be automatically recording all their work while they were at a terminal; this would be done in a non-intrusive manner and their participation (or non-participation) would have no effect whatsoever on their grade in the course. Student's participating in the study would receive the BUG-FINDER's analyses on only one assignment, the first looping program. (The problem for this program is given in Figure 10.)

Write a program which will input a set of numbers, where each number stands for the amount of rainfall in New Haven for a day. Compute the average rainfall, the number of rainy days, and the highest daily rainfall. Stop reading when 99999 is input; do not include this value in subsequent calculations. If the number input is negative, do not include it in the calculations and prompt the user to input another value.

Figure 10: The Noah Rainfall Problem: A First Looping Problem

Of the 900 students in the class approximately 116 volunteered to participate in the study. Needless to say the volume of data collected in this manner was quite substantial. The 116 students produced 970 *different* programs; in total, they produced 1504 programs, where 534 were the same.

We have analyzed, by hand, only a portion of that data. In particular, we have examined the first syntactically correct program produced by 20 randomly selected students who did receive the BUG-FINDER's analysis (see Table 1). Of the 99 we found in the 20 programs, the BUG-FINDER only correctly found 22 (22%). However, of the ones it found (40), it was correct 55% of the time; quite frankly, a success rate of 55% is not impressive. Clearly, the BUG-FINDER needed (1) to find more types of bugs, and it needed (2) to be more accurate.

7.2. Examples of Correct BUG-FINDER Analyses

In general, the BUG-FINDER was reasonably accurate in spotting simple assignment bugs, i.e., bugs in which a variable was assigned a value that was irrelevant, or bugs in which a variable was uninitialized. For example, consider User25's³ program fragment in Figure 11. We call this

³While the bugs described below were actually appeared in programs generated by students, the names of the students (e.g., User25) are fictitious.

Total Number of Bugs We Identified:	99
Bugs correctly reported by BUG-FINDER:	22 (22%)
Bugs incorrectly reported by BUG-FINDER:	18 (18%)
Bugs NOT reported by BUG-FINDER that should have been reported:	59 (60%)
Number of Bugs Reported by BUG-FINDER:	40
Correctly reported:	22 (55%)
Incorrectly reported:	18 (45%)
Number of Bugs NOT Reported by BUG-FINDER:	59
Bugs for which categories exist:	6 (15%)
Bugs for which categories do NOT exist:	53 (85%)

Table 1: Summary of Bug Statistics

```
27 AVERAGE := 0;
```

```
61 AVERAGE := SUM/VALID;
```

The BUG-FINDER responded by:

*** DOUBLE ASSIGNMENT BUG ***

AVERAGE GETS A VALUE IN THE ASSIGNMENT STATEMENT ON LINE 27,
BUT THEN BEFORE THIS VALUE IS EVER USED IT GETS A NEW VALUE
IN THE ASSIGNMENT STATEMENT ON LINE 61.

Figure 11: Code and Analysis of User25's Program

the DOUBLE ASSIGNMENT BUG since AVERAGE need not be initialized to zero. Note that strictly speaking, this program will execute; however, from an educational point of view, we feel that we need to point out that this type of coding practice is poor. Moreover, it might indicate a serious misconception surrounding assignment; clearly, studies with individual students are needed to evaluate this hypothesis. After receiving the BUG-FINDER's analysis, the student then took out the initialization of AVERAGE.

Similarly, consider User20's program in Figure 12. This program contained at least two bugs; the DOUBLE ASSIGNMENT BUG (RAINFALL need not be initialized), and the UNINITIALIZED VARIABLE BUG (RAINYDAYS was never set to zero). After seeing the BUG-FINDER's messages, the student then added a RAINYDAYS := 0, and took out the RAINFALL:=0.

Note that while both User20 and User25 had DOBULE ASSIGNMENT BUGs, it is distinctly possible that the underlying misconceptions in each case was different! That is, User25 may have some misconception about reading into a variable, while User20 may be confused about variables used in assignment statements. This simple example illustrates the difficulty of inferring misconceptions from program bugs.

Now consider the code in Figure 13 that appeared in User47's program. While this again illustrates the BUG-FINDER's ability to uncover a DOUBLE ASSIGNMENT BUG, what is interesting about this example is the student's response: after reading the BUG-FINDER's message, the student took out the *second* assignment. Again, interviewing the student as he was making this change might have shed some light on this curious -- though correct -- action.

Another common bug that the BUG-FINDER often accurately found, was the DIVIDE BY ZERO BUG; that is, if the first value read in was the sentinel value (99999), then the number of numbers read in would be equal to zero, and, in the average calculation a division by zero would cause a run-time error. For example, Figure 14 depicts a fragment from User77's program; that code appeared before code for writing out the value of AVEFALL. After receiving the BUG-FINDER'S analysis, the student then added a guard before the average calculation and corrected the bug.

The BUG-FINDER correctly identified an assortment of bugs in User55's program (Figure 15): DOUBLE ASSIGNMENT, UNINITIALIZED VARIABLE, DOUBLE ASSIGNMENT, INFINITE LOOP BUG, ADD IN BUG, and DIVIDE BY ZERO BUG. It took a several runs before the student corrected all the bugs in this program.

```
18  RAINFALL := 0;  
...  
23  READ( RAINFALL);  
...  
36  RAINYDAYS := RAINYDAYS+1
```

The BUG-FINDER responded by:

*** DOUBLE ASSIGNMENT BUG ***

RAINFALL GETS A VALUE IN THE ASSIGNMENT STATEMENT ON LINE 18,
BUT THEN BEFORE THIS VALUE IS EVER USED IT GETS A NEW VALUE
IN THE CALL TO READ ON LINE 23.

*** UNINITIALIZED VARIABLE BUG ***

THE ASSIGNMENT STATEMENT ON LINE 36 REFERS TO THE VALUE OF RAINYDAYS
BUT AT THIS POINT RAINYDAYS HASN'T GOTTEN A VALUE YET.

Figure 12: Code and Analysis of User25's Program

```
65 WHILE DAYRAIN<>SENTINEL DO
66     BEGIN
    ...
79     AVERAGE := TOTRAIN/DAYS
80     END;

83 AVERAGE := TOTRAIN / DAYS;
```

The BUG-FINDER responded by:

*** DOUBLE ASSIGNMENT BUG ***

AVERAGE GETS A VALUE IN THE ASSIGNMENT STATEMENT ON LINE 79,
BUT THEN BEFORE THIS VALUE IS EVER USED IT GETS A NEW VALUE
IN THE ASSIGNMENT STATEMENT ON LINE 83.

Figure 13: Code and Analysis of User47's Program

```
68 AVEFALL := TOTALFALL/TOTALCOUNT;
```

The BUG-FINDER responded by:

*** DIVIDE BY ZERO BUG ***

YOU HAVE NOT CORRECTLY CHECKED THAT TOTALCOUNT IS NOT ZERO
BEFORE DIVIDING BY TOTALCOUNT ON LINE 68.

YOUR PROGRAM WILL NOT HANDLE THE CASE WHEN NO VALID RAINFALLS ARE ENTERED.

Figure 14: Code and Analysis of User77's Program

```

10 RAINFALL = 0.
14 DAYS = 0.
24 READ( RAINFALL ).
25 WHILE RAINFALL<>SENTINEL DO
26   BEGIN
35     IF RAINFALL > HIGHRAIN THEN
36       HIGHRAIN = RAINFALL.
37     TOTALRAIN = TOTALRAIN+1
38     END.
40 DAYS = RAINYDAYS + DRYDAYS.
41 AVERAGE = TOTALRAIN / DAYS.

```

*** DOUBLE ASSIGNMENT BUG ***

RAINFALL GETS A VALUE IN THE ASSIGNMENT STATEMENT ON LINE 10.
BUT THEN BEFORE THIS VALUE IS EVER USED IT GETS A NEW VALUE
IN THE CALL TO READ ON LINE 24

*** UNINITIALIZED VARIABLE BUG ***

THE IF STATEMENT STARTING ON LINE 35 REFERS TO THE VALUE OF HIGHRAIN
BUT AT THIS POINT HIGHRAIN HASN'T GOTTEN A VALUE YET

*** DOUBLE ASSIGNMENT BUG ***

DAYS GETS A VALUE IN THE ASSIGNMENT STATEMENT ON LINE 14.
BUT THEN BEFORE THIS VALUE IS EVER USED IT GETS A NEW VALUE
IN THE ASSIGNMENT STATEMENT ON LINE 40

*** INFINITE LOOP BUG ***

THE WHILE LOOP STARTING ON LINE 25 IS AN INFINITE LOOP
THE VARIABLE RAINFALL IS NOT MODIFIED IN THE BODY OF THE LOOP.
SO IF THE CONDITION RAINFALL<>SENTINEL IS TRUE ON ENTERING THE LOOP
IT WILL NEVER BECOME FALSE. AND THE LOOP WILL EXECUTE FOR EVER

*** ADD IN BUG ***

IT SEEMS THAT YOU ARE TRYING TO ADD THE RAINFALLS INTO TOTALRAIN
BY INCREMENTING TOTALRAIN IN THE ASSIGNMENT STATEMENT ON LINE 37.
BUT THIS WILL NOT WORK

*** DIVIDE BY ZERO BUG ***

YOU HAVE NOT CORRECTLY CHECKED THAT DAYS IS NOT ZERO
BEFORE DIVIDING BY DAYS ON LINE 41
YOUR PROGRAM WILL NOT HANDLE THE CASE WHEN NO VALID RAINFALLS ARE ENTERED

*** 6 BUGS FOUND IN PROGRAM RAINFALL ***

Figure 15: Multiple Bugs Correctly Found

7.3. Examples of the BUG-FINDER'S Incorrect Analyses

Below, we list several cases in which the BUG-FINDER's analysis was incorrect. For example, Figure 16 depicts code from User90's program. In this particular case, when $\text{RAINFALL} = 0$ adding it into TOTAL has no effect. Thus, excluding it by the test of $\text{RAINFALL} > 0$ is quite acceptable. However, the BUG-FINDER was unable to deduce that $\text{RAINFALL} > 0$ and $\text{RAINFALL} \geq 0$ are equivalent in this case. The student changed the $>$ to $\geq$, and the BUG-FINDER was satisfied. While this appears to be a trivial bug to fix in the BUG-FINDER, it is indicative of the more general problem of coping with variability in student responses; while special purpose machinery can be built into a BUG-FINDER to reason about simple cases of equivalent test conditions, coping with the possibility of an infinite variety of ways of saying the same thing is quite another story! A mid-ground must be achieved, and this will be based primarily upon empirical considerations.

A more subtle incorrect analysis resulted from User23's program (Figure 17). The trouble with this program is that it is possible that 99999 could show up on the READ in the inner WHILE loop; thus, the IF test which guards the $\text{TOTLRNFL} := \text{TOTLRNFL} + \text{RAINFALL}$ line from the SENTINEL value is correct and needed. However, the BUG-FINDER thought that in the outermost WHILE loop the test for $\text{RAINFALL} \neq \text{SENTINEL}$ would be sufficient to protect the loop. Clearly, this program is complicated, and the dependencies and interactions between pieces of code are subtle. However, this type of program was seen quite often, and thus it can not be ignored.

User36's program (Figure 18a) is syntactically correct, however its structuring is somewhat confused. If we take the liberty of blocking this program in a more reasoned manner, the problem becomes more apparent (Figure 18b). Note that a BEGIN-END block is needed after the DO in the WHILE loop in order to enclose all the relevant code. Recognizing this omission is a difficult task; one would have to know what typically should go into the loop and be prepared to look around for that information. In Figure 18c we list the output from the BUG-FINDER --- which misses the point completely.

In Figure 19 we display a buggy program that highlights a key weakness in the current BUG-FINDER. Notice that it found a number of bugs (5); the problem is that these are all *local* bugs. Ignore for a moment the UNINITIALIZED VARIABLE BUG (bug #1) and the DIVIDE BY ZERO BUG (bug #5), which we feel are secondary problems with the program. The real problem is that the student has a loop, line 18 to 25, that reads in all the integers and discards the ones less than zero. The next loop, line 32 to 42, then processes all the valid rainfalls. Finally, the IF test (line 44) used to get at the maximum rainfall is also outside the loop. (Notice that the BUG-

BUGGY EXAMPLE 1

```
71  IF RAINFALL > 0 THEN
72      TOTAL := TOTAL + RAINFALL;
```

The BUG-FINDER responded by:

```
*** VALID INPUT BUG ***
YOU HAVE NOT MADE SURE THAT RAINFALL>=0 BEFORE
ADDING INTO TOTAL ON LINE 72 .
```

Figure 16: Code and Analysis of User90's Program

BUGGY EXAMPLE 2

```
39 WHILE RAINFALL <> SENTINEL DO
40   BEGIN
41
42     (*TEST FOR INVALID ENTRY*)
43     WHILE RAINFALL < 0 DO
44       BEGIN
45         WRITELN (RAINFALL, 'IS NOT A POSSIBLE RAINFALL');
46         WRITELN( 'TRY AGAIN');
47         READLN;
48         READ( RAINFALL );
49       END;
50
51     IF RAINFALL <> SENTINEL THEN BEGIN
52       TOTLRNFL := TOTLRNFL + RAINFALL;
53       ...
54       ...
66     READ(RAINFALL)
67   END;
```

The system responded by:

*** REDUNDANT TEST BUG ***

THE IF STATEMENT STARTING ON LINE 51 TESTS WHETHER RAINFALL<>SENTINEL, BUT IT IS ONLY POSSIBLE TO BE IN THE BODY OF THE WHILE LOOP STARTING ON LINE 39 WHEN THE CONDITION RAINFALL<>SENTINEL IS TRUE.

Figure 17: Code and Analysis of User23's Program

BUOXY EXAMPLE 3

(a)

-- Original Code User36)

```

34 WHILE RAINFALL <> 99999 DO
35
36   IF RAINFALL < 0 THEN
37     BEGIN
38       WRITELN('NEGATIVE RAINFALL IMPOSSIBLE. TRY AGAIN').
39       READLN
40       READ (RAINFALL)
41     END
42   ELSE
43     IF RAINFALL > 0 THEN
44       DAYS = DAYS + 1
45       IF RAINFALL > HIGHEST THEN
46         HIGHEST = RAINFALL
47       IF RAINFALL >= 0 THEN
48         BEGIN
49           TOTAL = TOTAL + RAINFALL
50           NUMBER = NUMBER + 1
51         READLN
52         READ ( RAINFALL)
53       END

```

(b)

-- User36's Program Restructured)

```

WHILE RAINFALL <> 99999 DO
  IF RAINFALL < 0 THEN BEGIN
    WRITELN('NEGATIVE RAINFALL IMPOSSIBLE. TRY AGAIN').
    READLN
    READ( RAINFALL )
  END
  ELSE IF RAINFALL > 0 THEN
    DAYS = DAYS+1
    HIGHEST = RAINFALL
    IF RAINFALL > HIGHEST THEN
      IF (RAINFALL >= 0) AND (RAINFALL <> 99999) THEN BEGIN
        TOTAL = TOTAL + RAINFALL
        NUMBER = NUMBER+1
        READLN
        READ( RAINFALL )
      END

```

(c)

-- BUG-FINDER's Analysis of User36's Program)

```

*** LAST ITERATION BUG ***
THE WHILE LOOP STARTING ON LINE 34 IS WRITTEN IN SUCH A WAY THAT
ON THE LAST ITERATION WHEN RAINFALL=99999 IT WILL STILL INCREMENT NUMBER
THEREFORE AFTER THE LOOP NUMBER WILL BE OFF BY 1
AND YOU WILL NOT GET THE CORRECT AVERAGE
YOU SHOULD USE A WHILE LOOP INSTEAD OF A REPEAT LOOP OR USE AN
IF STATEMENT INSIDE THE LOOP TO CHECK IF RAINFALL=99999 BEFORE INCREMENTING
NUMBER

```

Figure 18: Code and Analysis of User36's Program

FINDER failed to point out that the program does not find the maximum rainfall.) In a correct program, the first loop would be embedded in the loop at line 32, and the IF test would also be embedded in this loop. While the BUG-FINDER pointed out that the loop at line 32 is an infinite loop, this is only a symptom of the problem; the correction it suggests ignores the fact that the student did recognize that he needed to read in the values of RAINFALL — the student had a correct read loop starting at line 18. Moreover, bug #4, the VALID INPUT BUG, also ignores the fact that the student did have a loop that did filter out invalid (i.e., integers less than zero) input. The fact that there was a filter loop (line 18), a processing loop (line 32), and an IF test (line 44), all at the same level should have indicated to the BUG-FINDER the deeper misconception that was responsible for all three bugs. That is, apparently the student was trying to segment the task into 3 subtasks: filter the input, process the input, and search thru the input for the maximum. Conceptually that is precisely what the problem requires. From a global standpoint, the student's misconception is that he doesn't seem to be able to integrate pieces of code into a workable whole. Miller [12, 11] has observed bugs of this sort in his examination of non-programmers problem solving behavior.

Finally, the BUG-FINDER missed a serious bug in line 46; notice that the number of days in which there was no rainfall was not counted properly. `NORAIN := RAINFALL`, at line 46, does not count the non-rainy days. But notice that at line 47, the student has a valid counter update: `COUNT := COUNT + 1`. However, the student has already used COUNT to keep track of the total number of days (rainy and non-rainy). Moreover, while the indenting indicates that the COUNT update should be part of the IF body, there is no BEGIN-END block surrounding the statements. We saw a similar bug in BUGGY EXAMPLE 3 (Figure 18); we needed to infer that a BEGIN-END block was missing. If we assume that a BEGIN-END does wrap lines 46 and 47, the nature of the student's misconception becomes a bit clearer; apparently the student recognized that he needed to keep track the non-rainy days, but he was unable to implement that goal in code.

7.4. Why the BUG-FINDER Performed So Poorly

The BUG-FINDER'S database of bugs was built up primarily from an analysis of buggy programs novices generated for the problem in Figure 20a [20]. We thus tailored BUG-FINDER to find instances of those bugs in students' programs. However, for pedagogical reasons, the instructor of the class in which we tested BUG-FINDER insisted on giving the students a programming assignment which turned out to be more complex than we had intended. That is, the pencil and paper generation studies were carried out on a problem such as depicted in Figure 20a while BUG-FINDER was tested on the problem given in Figure 20b. The more complex

problem permitted the students a great deal more variability for implementing a correct solution --- and a great deal more freedom to make errors. The result, as we mentioned above, was that the effectiveness of BUG-FINDER was significantly lessened.

We did not appreciate the range of variability that was introduced when students were asked to do a harder programming assignment than was anticipated. Not only were our 18 types of bugs insufficient in magnitude, but more importantly, the number of ways that the same bug could crop up was significantly more than was expected. Moreover --- and here is the rub --- there were significant dependencies between pieces of code that had to be taken into account. The best example of this problem is in BUGGY EXAMPLE 4 (Figure 19); here we had to step back from the specific bugs and instead, draw a more coherent, total picture of the underlying misconceptions. In particular, we had to see that the program had all the pieces of the loop (more or less), and the problem was in integrating the pieces; because the BUG-FINDER was looking at local pieces of code, this global view could not be obtained. In other words, finding bugs can not be context independent.

Moreover, predictions about what should be in the code are needed in order to interpret pieces of code that would seem at first glance to be rather bizarre. For example, in BUGGY EXAMPLE 3 (Figure 18), we needed to hypothesize that a BEGIN-END should have surrounded pieces of code in order to make sense of the program. To make this hypothesis, we needed to know what the programmer might be intending to accomplish, i.e., we need to be able to make predictions about what should be in the code.

8. Concluding Remarks

Clearly, the MENO-II has quite limited functionality. We plan to significantly extend its capabilities along several dimensions.

1. The range of bugs and misconceptions which MENO-II is capable of coping with needs to be enlarged. The keys to this task are: (1) the continued development of a theory of programming knowledge [20, 21] and (2) empirical studies which seek to evaluate that theory and to identify additional bugs and misconceptions.
2. The ability to more accurately diagnose misconceptions is very important. Currently, only evidence from the program being analyzed is used in this process. We plan to have the TUTOR engage in a limited question-answer dialogue with the student in order to gather more information upon which to base its diagnosis. The student will not answer in natural language, but rather, will choose among a set of alternatives. Also, the history of the students' interactions will be kept and used in the analysis. By watching the pattern of bugs and patches that the student makes to the program, the system will be able to build up a better model of the student's understanding.

3. The question-answering interaction serves another purpose: it combines features of a Coaching strategy [7] with features of a Socratic Tutoring strategy [22]. That is, if one observes how a Program Consultant works with novice programmers who are having difficulties with their programs, one sees that oftentimes the students come to understand their misconceptions themselves; the Consultant's role was to ask carefully crafted questions which forced the students to confront their understanding of what they wanted to do with what they did in fact do. We feel that this paradigm of Consulting is one that is within the state-of-the-art reach of intelligent tutoring systems.

Our intention has been to build a CAI system with enhanced ability to understand and respond to students' buggy programs. In order to realize this objective, we have had to:

1. map out the deep structure knowledge used in programming
2. build a catalogue of common bugs
3. build a program analysis system which can find bugs in programs
4. associate misconceptions with bugs.

We have borrowed quite liberally from Artificial Intelligence and Cognitive Science in order to achieve these goals. Moreover, we feel that the time is ripe for just such limited AI-CAI systems --- systems with enhanced ability to understand and respond to the answers input by students. Personal computers with sufficient computing resources are becoming available, and the techniques for building AI-based systems are becoming less an art and more an engineering endeavor. Thus, we feel confident that the next generation of CAI systems will incorporate some AI techniques --- and will start appearing soon.

ACKNOWLEDGEMENTS

We would like to thank Bill Bregar for his encouragement and helpful comments on this paper, and we would also like to thank the reviewer for his helpful comments.

REFERENCES

1. Barstow, David. *Knowledge-Based Program Construction*. Elsevier North Holland Inc., 1979.
2. Bonar, J., Ehrlich, K., Soloway, E. "Collecting and Analyzing On-Line Protocols from Novice Programmers." *Behavioral Research Methods and Instrumentation* 14 (1982), 203-209.
3. Brachman, R., Ciccarelli, E., Greenfield, N., Yonke, M. KLONE Reference Manual. Tech. Rept. 3848, Bolt, Beranek and Newman, Inc, Cambridge, Mass., 1978.
4. Brown, J.S., Collins, A., and Harris, G. Artificial Intelligence and Learning Strategies. In H.O'Neil, Ed., *Learning Strategies*, Academic Press, New York, 1978.
5. Clancey, W.J., Bennett, J.S., and Cohen, P. Applications-oriented AI Research: Education. Tech. Rept. STAN-CS-79-749, Computer Science Department, Stanford University, 1979.
6. Goldstein, I.P. Understanding Simple Picture Programs. Tech. Rept. 294, MIT AI Lab, 1974.
7. Goldstein, I. The Computer as Coach: An Athletic Paradigm for Intellectual Education. Tech. Rept. 389, MIT AI Lab, Cambridge Ma., 1978.
8. Goldstein, I.P. "The Genetic Graph: A Representation for the Evolution of Procedural Knowledge." *International Journal of Man-Machine Studies* 1, 11 (1979), 51-78.
9. Johnson, L., Draper, S., Soloway, E. An Effective Bug Classification Scheme Must Take the Programmer into Account. SIGPLAN/SIGSOFT Workshop on High-Level Debugging, in press.
10. Johnson, L., Draper, S., Soloway, E. The Nature of Bugs in Novices' Pascal Programs. in preparation
11. Miller, L. A. "Natural Language Programming: Styles, Strategies, and Contrasts." *IBM Systems Journal* 20 (1981), 184-215.
12. Miller, L.A. "Programming by Non-Programmers." *International Journal of Man-Machine Studies* 6 (1974), 237-260.
13. Miller, Mark L. "A Structured Planning and Debugging Environment for Elementary Programming." *Int. J. Man-Machine Studies* 11 (1978), 79-95.
14. Miller, J. R., Kehler, T. P., Michaelis, P. R., & Murray, W. R. . Intelligent Tutoring for Programming Tasks: Using Task Analysis to Generate Better Hints. Tech. Rept. TI ONR-TR-82-0818F, Texas Instruments, Inc., 1982.
15. Rich, C. Inspection Methods in Programming. Tech. Rept. AI-TR-604, MIT AI Lab, 1981.
16. Rich, C. and Shrobe, H. "Initial Report on a LISP Programmer's Apprentice." *IEEE Transactions on Software Engineering* 4, 6 (November 1978), 342-376.

17. Soloway, E. and Woolf, B. Problems, Plans, and Programs. Proceedings of Eleventh ACM Technical Symposium on Computer Science Education, ACM, 1979.
18. Soloway, E., Bonar, J., Woolf, B., Barth, P., Rubin, E., and Ehrlich, K. Cognition and programming: Why Your Students Write Those Crazy Programs. Proceedings of the National Educational Computing Conference, NECC, No. Denton, Tx., 1981.
19. Soloway, E., Bonar, J., Ehrlich, K. . Cognitive Strategies and Looping Constructs: An Empirical Study. Communications of the ACM, in press.
20. Soloway, E., Ehrlich, K., Bonar, J., Greenspan, J. What Do Novices Know About Programming? In A. Badre, B. Shneiderman, Ed., *Directions in Human-Computer Interactions*, Ablex, Inc., 1982.
21. Soloway, E., Ehrlich, K., Bonar, J. Tapping Into Tacit Programming Knowledge. Proceedings of the Conference on Human Factors in Computing Systems, NBS, Gaithersburg, Md., 1982.
22. Stevens, A. and Collins, A. The Goal Structure of a Socratic Tutor. Bolt Beranek and Newman, Cambridge, Mass., 1977.
23. Teitelbaum, T. and Reps, T. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. Tech. Rept. 80-421, Cornell University, Dept. of Computer Science, 1980.
24. Waters, R.C. "A Method for Analyzing Loop Programs." *IEEE Trans. on Software Engineering* SE-5 (May 1979), 237-247.
25. Westcourt, K.T., Beard, J. Gould, L. and Barr, A. Knowledge-Based CAI: CINS for Individualized Curriculum Sequencing. Tech. Rept. Technical Report 290, Institute for Mathematical Studies in the Social Sciences, Stanford, 1977.

BUGGY EXAMPLE 4

```

1 PROGRAM AVERRAIN (INPUT/ OUTPUT)
2
3 CONST
4   SENTINEL = 99999
5
6 VAR
7   RAINFALL AVERRAIN HIGHRAIN NORAIN REAL
8   TOTALRAIN VALIDRAINS RAINYDAYS COUNT REAL
9
10 BEGIN
11   WRITELN ('HI NOAH, THIS IS A PROGRAM THAT COMPUTES THE AVERAGE').
12   WRITELN ('RAINFALL, HIGHEST RAINFALL, NUMBER OF RAINFALL DAYS').
13   WRITELN ('ENTERED, AND THE TOTAL NUMBER OF RAINYDAYS ENTERED').
14   WRITELN ('ENTER YOUR RAINFALL DATA NOW AND TRY TO REMEMBER THAT').
15   WRITELN ('YOU CAN NOT ENTER NEGATIVE INPUT').
16
17 READLN
18   READ (RAINFALL)
19   WHILE RAINFALL < 0 DO
20     BEGIN
21       WRITELN ('NOAH YOU IDIOT, YOU ENTERED NEGATIVE RAINFALL').
22       WRITELN ('YOU CANT HAVE NEGATIVE RAINFALL NOW ENTER ').
23       WRITELN ('CORRECT RAINFALL AGAIN NOAH ').
24       READ (RAINFALL)
25     END
26
27   WRITE ('THE RAINFALL ENTERED WAS' RAINFALL 0 2, 'INCHES').
28   COUNT = 0
29   HIGHRAIN = 0
30   TOTALRAIN = 0
31
32   WHILE RAINFALL <> SENTINEL DO
33     BEGIN
34       IF RAINFALL > HIGHRAIN
35       THEN
36         HIGHRAIN = RAINFALL
37         TOTALRAIN = TOTALRAIN + RAINFALL
38         COUNT = COUNT + 1
39         AVERRAIN = TOTALRAIN / COUNT
40         VALIDRAINS = COUNT - NORAIN
41         RAINYDAYS = VALIDRAINS
42     END
43
44   IF RAINFALL = 0
45   THEN
46     NORAIN = RAINFALL
47     COUNT = COUNT + 1
48
49   WRITELN ('THE NUMBER OF VALIDRAIN DAYS ENTERED WAS', VALIDRAINS 0 2).
50   WRITELN ('DAYS')
51   WRITELN ('THE AVERRAIN WAS' AVERRAIN 0 2, 'INCHES PER DAY').
52   WRITELN ('THE HIGHRAIN IS' HIGHRAIN 0 2, 'INCHES').
53   WRITELN ('THE NUMBER OF RAINYDAYS IN THIS PERIOD WAS').
54   WRITELN (RAINYDAYS 0 2, 'DAYS').
55 END

```

*** UNINITIALIZED VARIABLE BUG ***

THE ASSIGNMENT STATEMENT ON LINE 40 REFERS TO THE VALUE OF NORAIN BUT AT THIS POINT NORAIN HASN'T GOTTEN A VALUE YET

*** INFINITE LOOP BUG ***

THE WHILE LOOP STARTING ON LINE 32 IS AN INFINITE LOOP. THE VARIABLE RAINFALL IS NOT MODIFIED IN THE BODY OF THE LOOP. SO IF THE CONDITION RAINFALL<>SENTINEL IS TRUE ON ENTERING THE LOOP IT WILL NEVER BECOME FALSE, AND THE LOOP WILL EXECUTE FOR EVER

*** LAST ITERATION BUG ***

THE WHILE LOOP STARTING ON LINE 32 IS WRITTEN IN SUCH A WAY THAT ON THE LAST ITERATION WHEN RAINFALL=99999 IT WILL STILL INCREMENT COUNT. THEREFORE AFTER THE LOOP COUNT WILL BE OFF BY 1 AND YOU WILL NOT GET THE CORRECT AVERAGE. YOU SHOULD USE A WHILE LOOP INSTEAD OF A REPEAT LOOP, OR USE AN IF STATEMENT INSIDE THE LOOP TO CHECK IF RAINFALL=99999 BEFORE INCREMENTING COUNT

*** VALID INPUT BUG ***

YOU HAVE NOT CORRECTLY CHECKED THAT RAINFALL IS POSITIVE BEFORE ADDING RAINFALL INTO TOTALRAIN ON LINE 37

*** DIVIDE BY ZERO BUG ***

YOU HAVE NOT CORRECTLY CHECKED THAT COUNT IS NOT ZERO BEFORE DIVIDING BY COUNT ON LINE 39. YOUR PROGRAM WILL NOT HANDLE THE CASE WHEN NO VALID RAINFALLS ARE ENTERED

Figure 19: Archtypical Example of a Complicated Buggy Program

A Simple Looping Problem:

(a)

Read in a set of integers and print out their average. Stop reading numbers when the number 99999 is seen. Do NOT include the 99999 in the average.

(b)

A More Complex Problem:

Write a program which will input a set of numbers, where each number stands for the amount of rainfall in New Haven for a day. Compute the average rainfall, the number of rainy days, and the highest daily rainfall. Stop reading when 99999 is input; do not include this value in subsequent calculations. If the number input is negative, do not include it in the calculations and prompt the user to input another value.

Figure 20: Simple and Complex Problems

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER #258	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MENO-II: An AI-Based Programming Tutor		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER #258
7. AUTHOR(s) Elliot Soloway, Eric Rubin, Beverly Woolf, Jeffrey Bonar, W. Lewis Johnson		8. CONTRACT OR GRANT NUMBER(s) N00014-82-K-0714
9. PERFORMING ORGANIZATION NAME AND ADDRESS Yale University 10 Hillhouse Avenue New Haven, CT 06520		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 154-492
11. CONTROLLING OFFICE NAME AND ADDRESS Personnel and Training Research Programs Office of Naval Research (Code 458) Arlington, VA 22217		12. REPORT DATE August 1983
		13. NUMBER OF PAGES 35
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES To appear: The Journal of Computer Based Instructional Systems		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Artificial Intelligence, Expert System, Automatic Program Understanding, Debugging Aids, Programming Plans, Tutoring Systems		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) MENO-II is a computer-based tutor intended to help novices learning to program in Pascal. The BUG-FINDING component attempts to find non-syntactic bugs in a student's program. It draws on a database of 18 common bug types, represented as templates, and attempts to match these templates against its analysis of the student's program. The TUTORING component then attempts to infer the misconception that might underlie the bug and present the student with remedial instruction. We tested the BUG-FINDING component in a classroom setting: we report here on an analysis of the system's performance.		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

5 N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)